Poor Man's Compilers

how Forth treats its source code

Klaus Schleisiek

kschleisiek at freenet.de

Topics

- How Forth came about
- The Forth System
- The Dictionary
- Parsing in Forth
- Compilation in Forth
- Syntactic Sugar

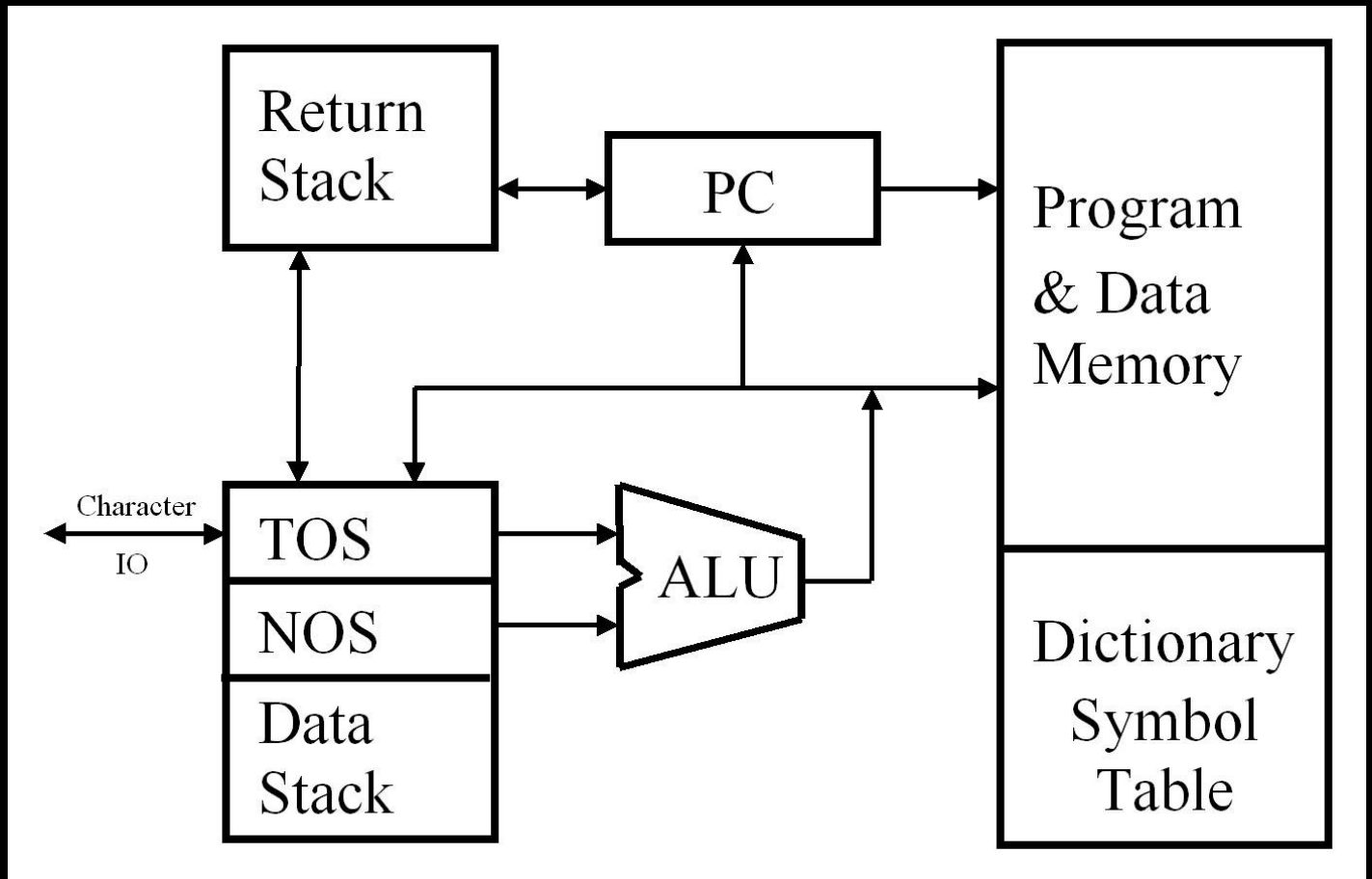
How Forth came about

Developed between late 1950 and late 1960 by Chuck Moore as a byproduct of his work as a freelance programmer of automation systems, which often had only an assembler available for the software

Chuck Moore: "I developed Forth over a period of some years as an interface between me and the computers I programmed. The traditional languages were not providing the power, ease, or flexibility that I wanted."

The Forth System

Forth is based on a simple virtual machine. It is the assembler of this machine:



The Dictionary

This is what Forth calls its symbol table.

The dictionary is a tree structure of word lists (vocabularies) that remain available after compilation.

Identical names can have different semantics in different vocabularies.

Context is the name of the dynamically changeable list of vocabularies that are searched to identify a token.

Current is the name of the vocabulary into which new word definitions are entered.

Parsing in Forth

The lexical analysis looks only for whitespace (blank, tab, cr, etc.) to isolate tokens. Therefore, names in Forth can contain any ASCII character except whitespace.

The parser searches Context to see if the token exists there.

- If yes, the corresponding code is executed.
- If no, it tries to interpret the token as a number. If successful, the number is put on the stack.
- If neither, Forth responds with ?.

This is the system behavior when the source code is read in interpretation state.

Compilation in Forth

The Colon compiler puts the system into compilation state:

```
: xlerb  word1 word2 word3 ;
```

creates the word entry xlerb in Current, whose semantics is defined by **word1** | **word2** | **word3** executed in sequence.

Other predefined compilers are:

```
Variable <varname>

<number> Constant <constname>

Vocabulary <vocname>
```

The Immediate Bit

It is a marker bit in the word name of a vocabulary entry. If it is set, this word will be executed immediately also in compilation state (see: ;) and not compiled as a function call.

This can be used to create control structures:

```
: conditional ( flag -- )
   IF  word4  ENDIF  word5 ;
```

IF is immediate, compiles 0=branch and puts the address of the following memory cell on the stack (during compilation!), which will later hold the target address of the branch. Initially, this memory cell is usually filled with 0.

ENDIF is also immediate and stores the target address at the address which was put on the stack by IF.

Compilation in Forth

In Forth there is hardly any predefined syntax and no global compiler that checks the grammar rules of the source code.

Instead, there are individual small compilers that interact synergistically to generate syntax.

Additional compilers can be defined as part of an application.

The syntax of Forth is dynamically extensible.

Syntactic Sugar

ASCII text is to be output as Morse tones. For this you need a code table. This is error-prone and should therefore be written as follows:

```
morsetable:
  . _      | A       _ .      | N
  _ . . .  | B       _ _ _    | O
  _ . _ .  | C       . _ _ .  | P
  _ . .    | D       _ _ . _  | Q
  .        | E       . _ .    | R
  . . _ .  | F       . . .    | S
  and so on ...
;morsetable
```

Which compilers are required to realize this syntax?

Syntactic Sugar

```
$80 cells Constant #codes

Create Morsetable #codes allot   Morsetable #codes erase

Vocabulary <morse>

: morsetable: ( -- 0 )
   also <morse>           \ make <morse> first vocab. in CONTEXT
   0                      \ end 'marker' for the first code
;

also <morse> definitions \ set CURRENT to <morse>

1 Constant .

2 Constant _
```

Syntactic Sugar

```
: morsecode! ( count bits <character> -- )
   swap 8 lshift or     \ pack count and code into 16 bit value
   char cells           \ offset into morse table
   dup #codes >= abort" character out of range"
   Morsetable + !       \ store at char's position
;
: | ( 0 n1 .. nr -- 0 )
   0 ( count ) 0 ( morsebits )
   BEGIN rot ?dup
   WHILE 1- swap 2* or  \ add next morsebit
         swap 1+ swap   \ and increment count
   REPEAT morsecode!
   0                    \ end marker for the next code
;
: ;morsetable ( 0 -- )
   abort" malformed morse table"  \ 0 must be on the stack
   previous                       \ remove <morse> from CONTEXT
;
previous definitions    \ reset CURRENT to previous vocabulary
```